MTE Project

Graph Theory

Graph Theory based Beam Search to find Nearest Neighbour in d-Dimensional Metric Space

Kunal Sharma
2K18/MC/060

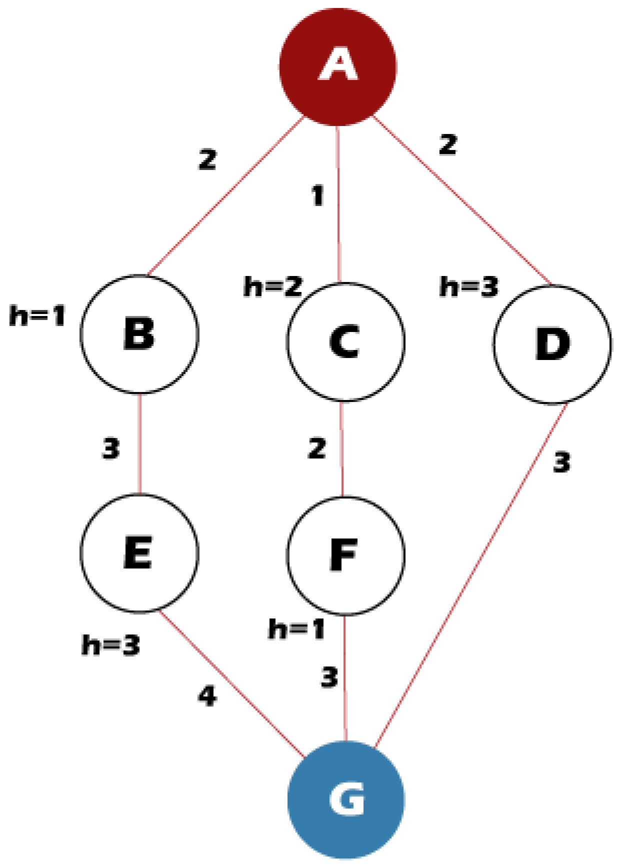
Janardan Upadhyay
2K18/MC/047

Department of Applied Mathematics

Submitted To:- Prof Sangita Kansal

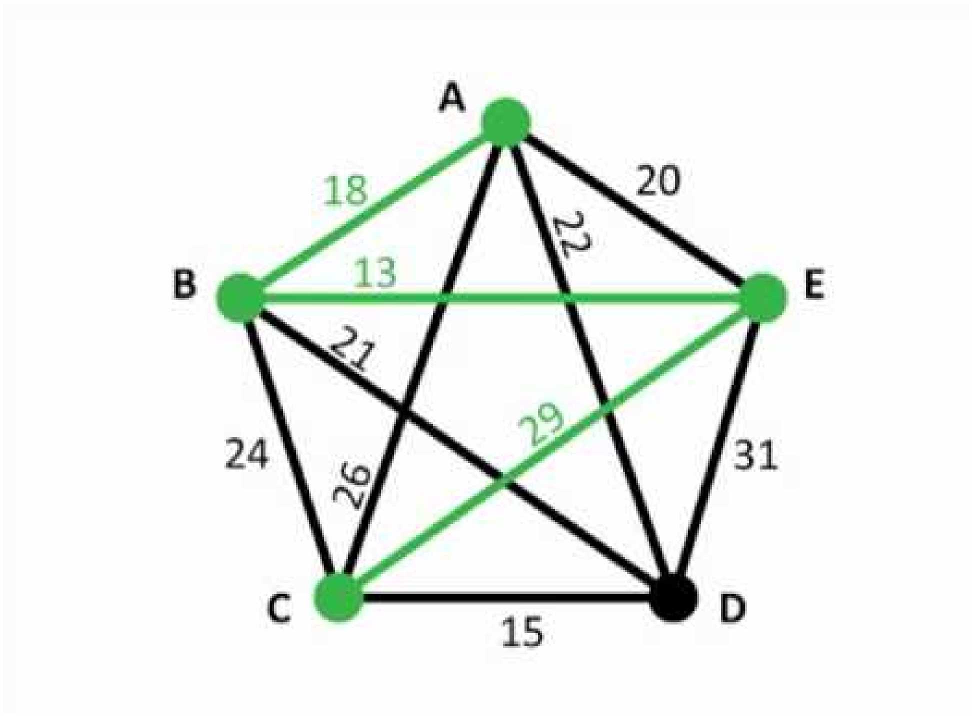
Index

Abstract



Graph Constructed Nearest Neighbour Search has been very successful for finding some of the neighbourhoods of a particular query in a dataset or even for finding the k-Nearest Neighbours as in kNN Algorithm. The NN algorithms have been able to work pretty neatly on statistically sound synthetic data from which we can create graphs. To implement these complex Graph and Tree searching algorithms we find the lack of working methods on real datasets that we can create. We in this project are trying to create a new methodology for graphs searching algorithms to work on low-dimensional ($d << \log n$) datasets so that we can use state of the art search algorithms for artificially created real data. Majorly we will be working with Beam Search algorithm and Beam Stack Search Algorithm on graphs for NNS finding and reporting work via maintaining a dynamic list of NN candidates of graph.

Introduction

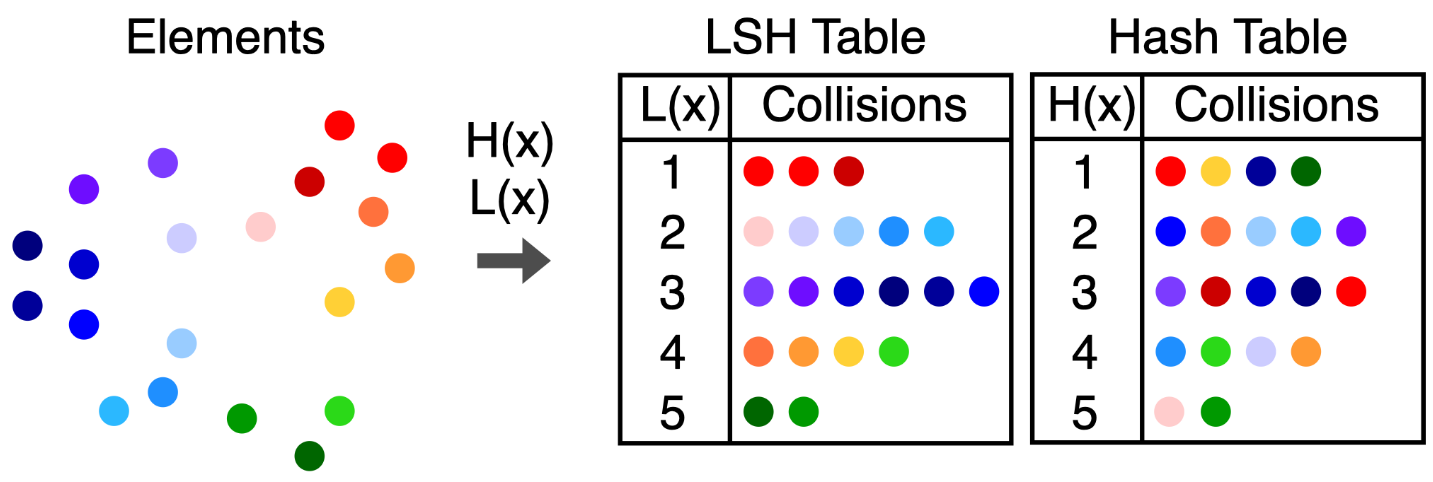


1. Theoretical Foundations of NN algorithms and their needs in Graph:

A. Nearest neighbour search is used in many approaches in machine learning, pattern recognition, coding theory, and other fields of research. The k-nearest neighbour technique, in particular, is listed among the top 10 data mining methods. Given the size of today's datasets (both in terms of the number of items n and the dimension d), lowering the computational cost of NNS methods is critical. The closest neighbour issue is to preprocess a given dataset D in such a manner that we can rapidly (in time o(n)) locate its nearest neighbours in D for any arbitrary upcoming query vector q.

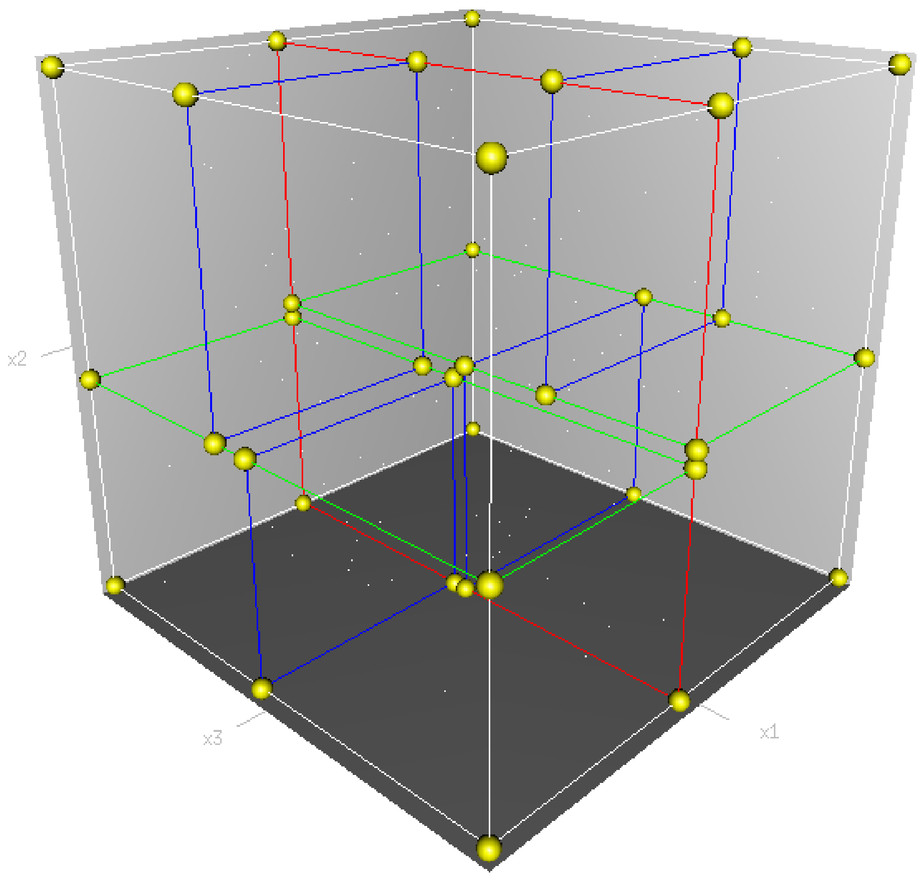
B. For the NN problem, there are several efficient solutions, especially when the dimension d is small. Algorithms based on recursive partitions of the space, such as k-d threes and random projection trees, are particularly popular. The Locality Sensitive Hashing (LSH) method, which is thoroughly researched theoretically and frequently utilised in practical applications, is the most well-known algorithm for large d.

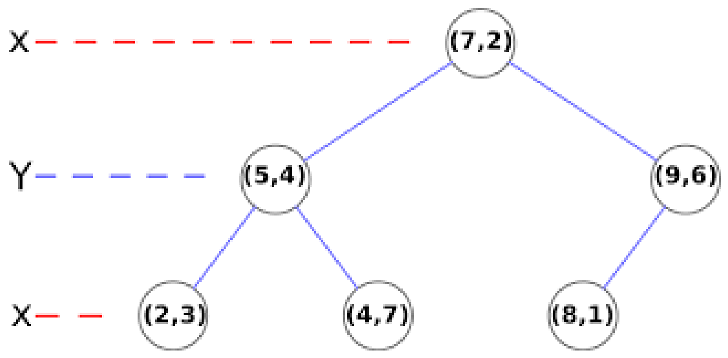
C. In a number of large-scale NNS applications, graph-based methods have recently been demonstrated to outperform other types of algorithms. The majority of graph-based techniques work by building a nearest neighbour graph (or its approximation), in which nodes correspond to D items and each node is connected to its nearest neighbours through directed edges. Then, for a given query q, one selects an element from D (either random or preset) and takes greedy steps on the network towards q: at each step, all neighbours of a current node are assessed, and the one closest to q is picked. To speed up graph-based search, a number of new heuristics have been suggested.



D. Our analysis in this project is dependent on the uniform distribution of synthetic data on a sphere and focusing on the dense region of the data on (d << log n), uniformization and densification applied to any particular dataset is going to improve the pre-applied NNS Algorithms. We start by looking at how greedy search works on basic NN networks in both dense and sparse regimes. For some constant M, the temporal complexity in the dense regime is Θ (exp(n, 1/d)*M.d). Here M.d corresponds to the complexity of one step and exp(n, 1/d) to the number of steps.

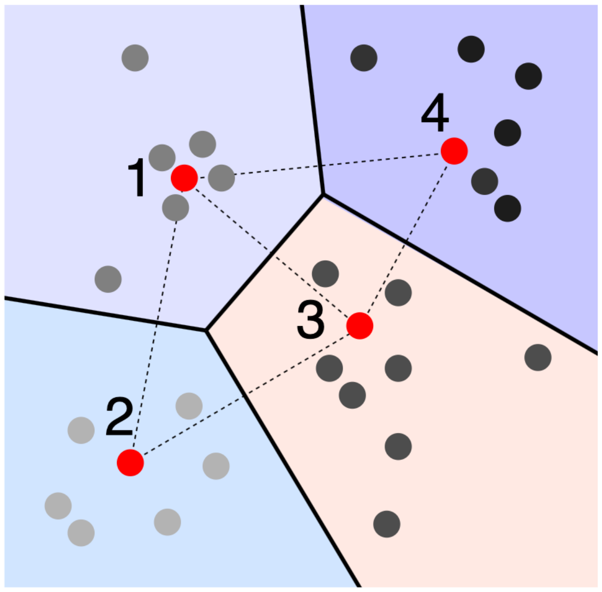
E. We also take into account a heuristic that has been shown to increase accuracy: keeping a dynamic list of multiple possibilities rather than just one ideal location, which is a generic approach known as beam search. The impact of this method is thoroughly investigated.





2. Graph based Nearest Neighbour algorithms for searching:

A. Several well-known NNS algorithms, such as k-d trees and random projection trees, are based on recursive partitions of the space. A k-d tree's query time is O(d2O(d)), resulting in a fast method for the precise NNS in the dense regime log n. Because all methods suffer from the dimensionality curse when dlog n, the issue is frequently simplified to approximate nearest neighbour.

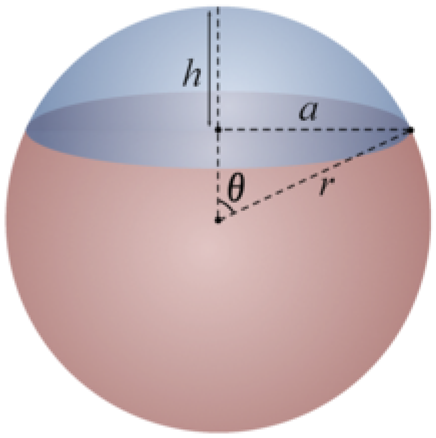


B. LSH is the most theoretically researched of the methods developed for the ANN issue. The primary idea behind LSH is to hash the points so that the likelihood of colliding is considerably higher for points that are close together than for ones that are far apart. Then, by hashing the query and verifying the entries in the same buckets, one may obtain close neighbours for it. LSH solves c-ANN with query timeΘ (dnφ) and space complexity Θ(n1+φ).

C. When the dimension d is logarithmic in the number of components n, LSF can outperform LSH. LSF is a technique that falls in between LSH and graph-based approaches. LSF defines filters with tiny spherical caps, but graph-based techniques enable, generally speaking, shifting to nearby caps.

Methodology

1. Setup and Notations



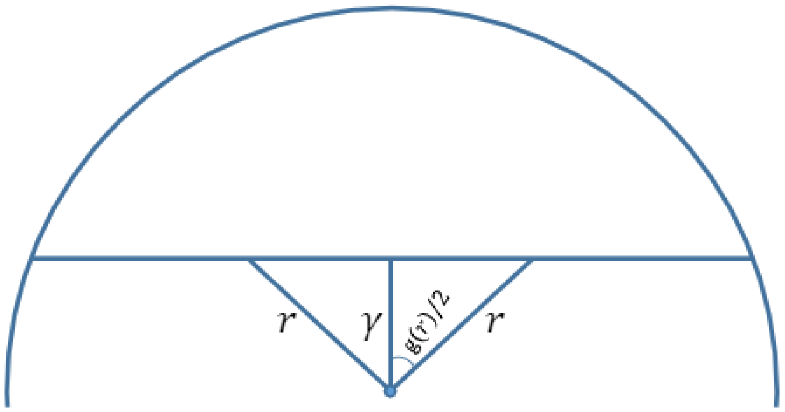
A. We're given the dataset D={x1,...xn,xi} Rd+1, and we're told that all of the components of D correspond to a unit Euclidean sphere, D is a subset of d-Sphere. Because feature vectors are frequently normalised, this specific situation is very important for practical applications. Let x'' D be the nearest neighbour of a given query q in d-Sphere. The goal of the exact NNS is to return x', whereas in c, R-ANN (approximate near neighbour), we need to discover such x′ that (q, x′) < cR if(q, x) < R for given R >0,c >1. We also express a spherical distance with (,).

B. As a result, it is a critical step in comprehending the limitations and advantages of graph-based NNS algorithms. Real datasets are rarely evenly distributed from a practical standpoint. However, uniformity is useful in some applications, and there are ways for making a dataset more uniform while keeping the distances as close as possible

C. Most graph-based approaches, as mentioned in the introduction, are focused on creating a closest neighbour graph (or its approximation). Linking an element x to a certain number of nearest neighbours for uniformly distributed datasets is essentially similar to connecting it to all such nodes y that (x, y) with some suitable.

D. As a result, during the preprocessing step, we select some and use this threshold to create a graph. When we obtain a query q, we choose a random element x D so that (x, q)/2 and execute a graph-based greedy descent: at each step, we estimate the distances between the neighbours of a current node and q and advance to the closest neighbour.

2. Theoretical Analysis and Auxiliary Results of corresponding theorem..



$$C(\gamma) = \frac{(d-1)\,\hat{\gamma}^{d-1}}{2\,\pi} \int_0^1 \arccos\left(\frac{\gamma}{\sqrt{1-\hat{\gamma}^2 t}}\right) t^{\frac{d-3}{2}}\,dt$$

$$= \frac{(d-1)\,\hat{\gamma}^{d-1}}{2\,\pi} \int_0^1 \arcsin\left(\hat{\gamma}\sqrt{\frac{1-t}{1-\hat{\gamma}^2 t}}\right) t^{(d-3)/2}\,dt$$

Now we note that $x \leq \arcsin(x) \leq x \cdot \pi/2$ for $0 \leq x \leq 1$, so

$$C(\gamma) = \Theta\,(d)\,\hat{\gamma}^d \int_0^1 \sqrt{\frac{1-t}{1-\hat{\gamma}^2 t}} \cdot t^{(d-3)/2}\,dt\,.$$

Finally, we estimate

$$\sqrt{1-t} \leq \sqrt{\frac{1-t}{1-\hat{\gamma}^2 t}} \leq \min\left\{1, \sqrt{\frac{1-t}{1-\hat{\gamma}^2}}\right\}.$$

So, the lower bound is

$$C(\gamma) \geq \Theta\,(d)\,\hat{\gamma}^d\,\mathrm{B}\left(\frac{3}{2}, \frac{d-1}{2}\right) = \Theta\,(d)\,\hat{\gamma}^d \left(\frac{d-1}{2}\right)^{-3/2} = \Theta\left(d^{-1/2}\right)\hat{\gamma}^d\,.$$

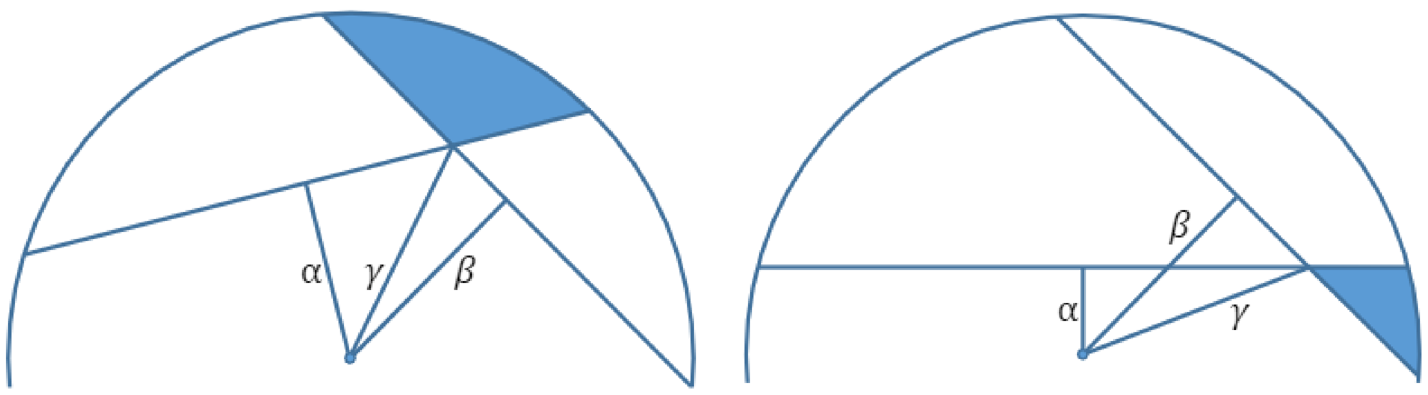
The upper bounds are

$$C(\gamma) \leq \Theta\,(d)\,\hat{\gamma}^d \int_0^1 t^{(d-3)/2}\,dt = \Theta\,(1)\,\hat{\gamma}^d\,,$$

$$C(\gamma) \leq \Theta\,(d)\,\hat{\gamma}^d \int_0^1 \sqrt{\frac{1-t}{1-\hat{\gamma}^2}} \cdot t^{(d-3)/2}\,dt = \Theta\left(d^{-1/2}\right)\frac{\hat{\gamma}^d}{\gamma}\,.$$

This completes the proof.

A. The first time for creating and modifying real dataset for construction of graphs is the construction of unit Euclidean Spheres where distance is measured along the great circle.



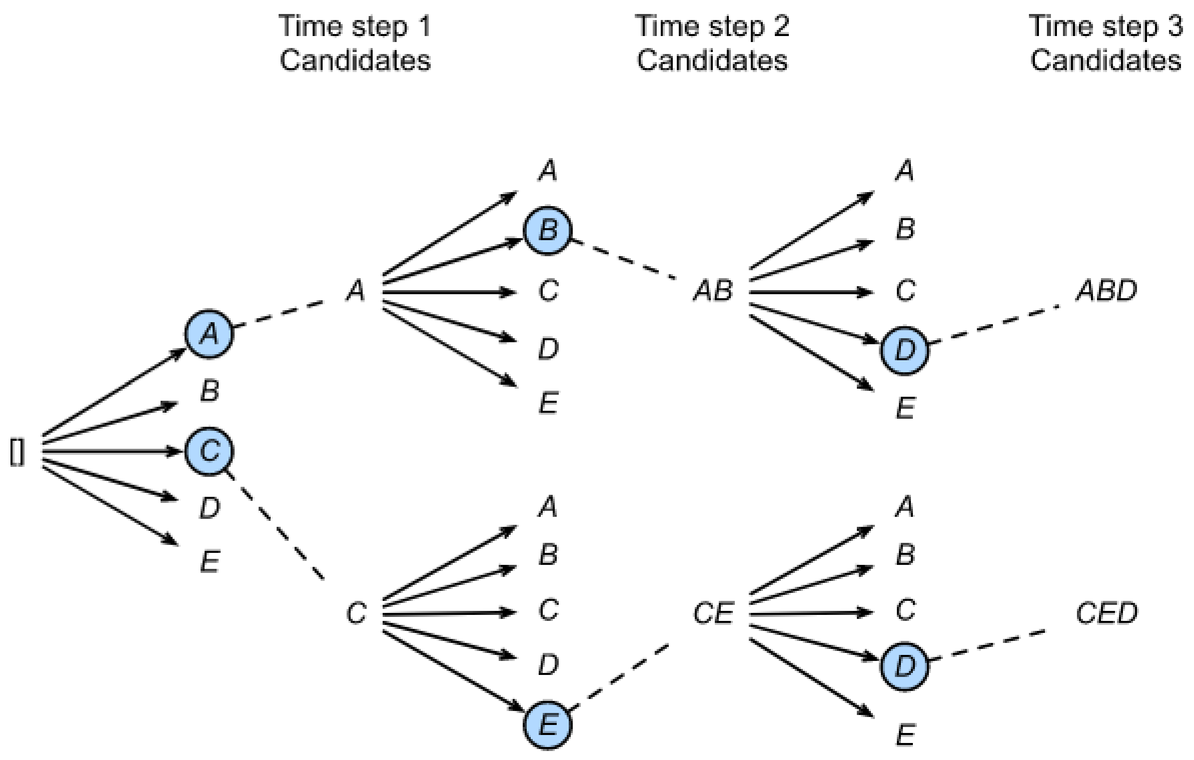
$$\mathrm{P}(\text{edge from } u \text{ to } v) = \frac{\rho(u,v)^{-d}}{\sum_{w \neq u} \rho(u,w)^{-d}}\,.$$

$$\mathrm{P}(\text{edge to } k\text{-th neighbor}) = \frac{1/k}{\sum_{i=1}^{n} 1/i} \sim \frac{1}{k \ln n}\,.$$

B. Greedy Algorithms on Plain Graphs for searching Nearest Neighbours, according to theorem for Dense regime := Let G(M) be a graph formed by connecting xi and xj iff (xi, xj) arcsin(M*exp(n, -1/d).

C. Using Long-range Links - Making the diameter of the Graph G(M) smaller.
The number of steps is minimal in comparison to the one-step difficulty when d >> sqrt(log n). As a result, lowering the number of steps has no effect on the asymptotics' primary term. If d << sqrt(log n) (very dense setup), however, the number of steps becomes the most important factor in time complexity. In this scenario, reducing the number of steps by introducing so-called long-range connections (or shortcuts) — certain edges linking components that are far apart — may speed up the search in the early stages of the algorithm.

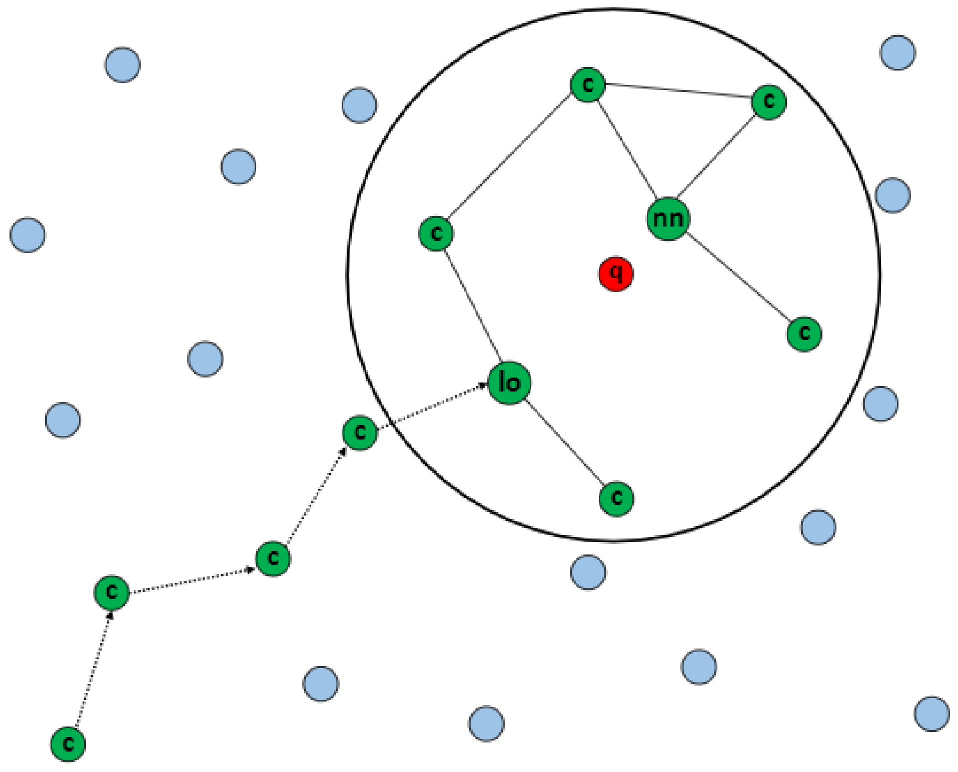
D. Beam Search



Beam search is a heuristic technique that expands the most promising element in a constrained collection to explore a graph. It is commonly used in graph-based NNS algorithms since it enhances accuracy significantly. Furthermore, it was demonstrated that on synthetic datasets, even a basic kNN graph combined with beam search may produce results that are near to the state-of-the-art. However, to our knowledge, we are the first to investigate the theoretical effects of beam search in graph-based NNS algorithms.

E. Let us demonstrate how M can be decreased by using beam search instead of greedy search. Assume we've arrived at a local neighbourhood of the query q, and there are m1 points in the dataset that are closer to q. Assume we're doing beam search and have at least m top candidates saved. If the subgraph on the m closest nodes to q is linked, the beam search finishes after at most m steps and provides the m closest nodes, including the nearest neighbour. As a result, we can shrink the neighbourhood, but rather than becoming trapped in a local optimum, we explore the area until we achieve the objective. The following theorem holds in a formal sense.

Implementation

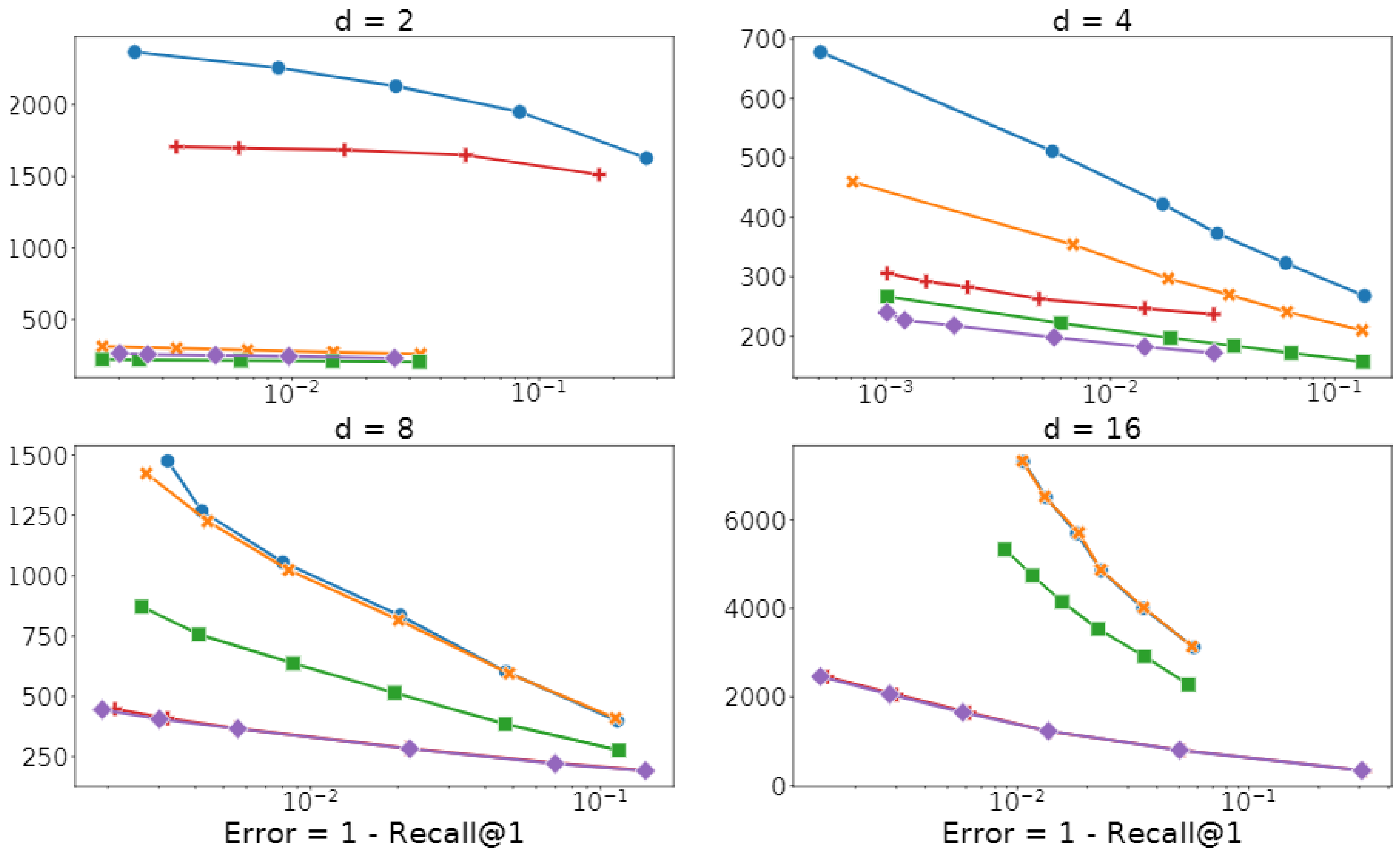


A. Experimental Terms

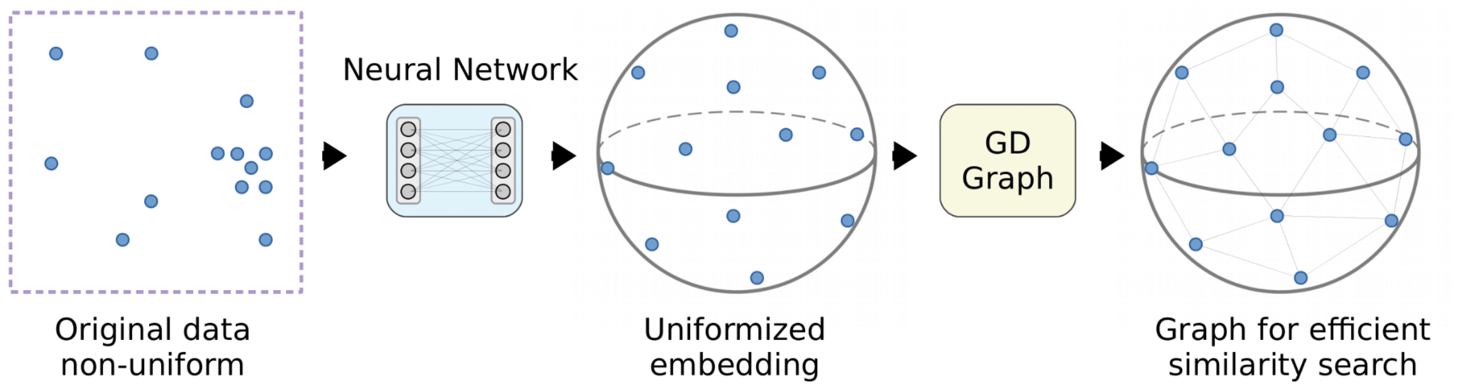
kNN - Simple NN Graph, where each node has a fixed number of neighbours. We will additionally add kL (which means pre-sampling for the database is done). At each iteration we are going to use Long Range Link to shorten the diameter of the graph under consideration. Then finally the BEAM method is used for faster computation.

B. From previous Research on this topic what we can learn is that Graph based Beam Searching works pretty well on Synthetic Dataset, some of the results are shown below. But in Real data or data that is not statistically neat, we need additional functions and the concept of DIM-RED which is **Dimensionality Reduction** of data to make it appear as the real data for dense systems (d << log n) on graphs.

C. Results obtained during previous research on Real Dataset



D. Now for Real we cannot create so many points as to make log(n) >> d, hence we might need to reduce d more than we need to increase n. Hence we use the concept of Dim-Red.


Neural Network
GD Graph
Original data non-uniform
Uniformized embedding
Graph for efficient similarity search

E. Python Code for Dimensionality Reduction and reporting differences in Graph.

```
from __future__ import division
import time
import numpy as np
from torch import nn, optim
import torch.nn.functional as F
import torch
import itertools

from dim_red.support_func import  forward_pass, Normalize, stopping_time,\
    repeat, get_nearestneighbors_partly, save_transformed_data, ifelse,\
    validation_function, save_net_as_matrix

from dim_red.data import write_fvecs

net_style = "angular"


def get_graph_dot_prod(graph_cur, xt_var):
    graph_directions = graph_cur - xt_var.reshape(graph_cur.shape[0], 1,
graph_cur.shape[-1])
    graph_directions = graph_directions / (graph_directions.norm(dim=-1, keepdim=True) +
0.00001)
    dot_prod = torch.bmm(graph_directions, graph_directions.permute(0, 2, 1))

    return dot_prod
```

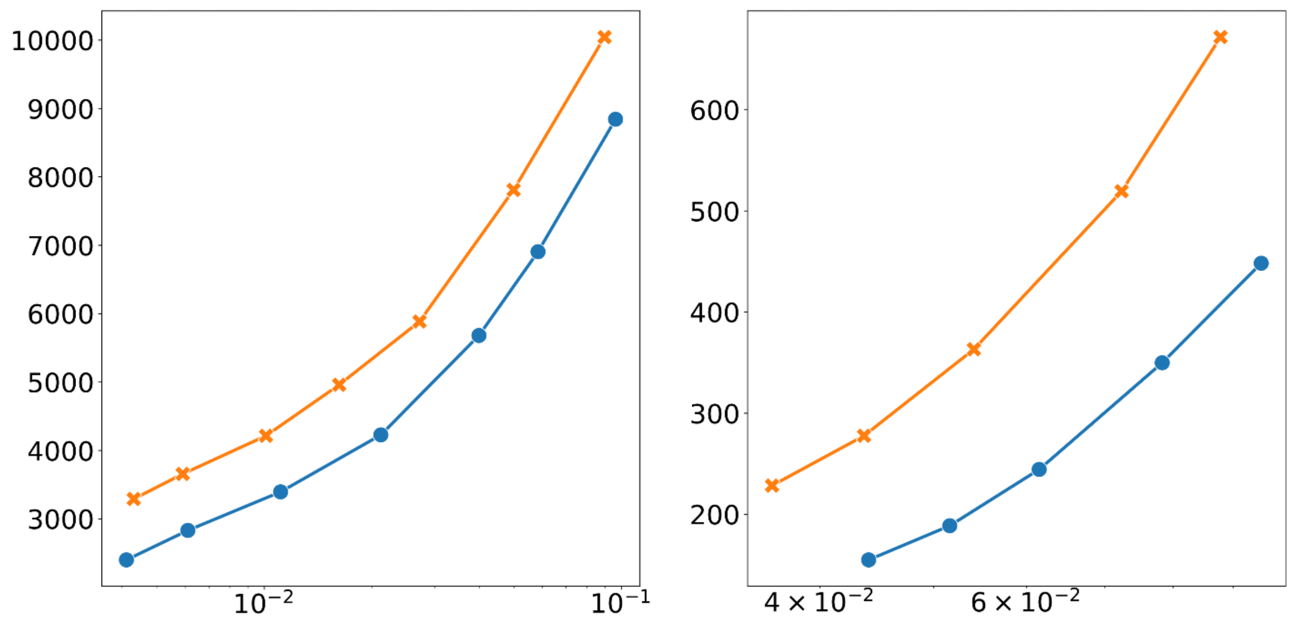
```
def angular_loss(graph, xt_var, net, bn, args):
    x_dot_prod = get_graph_dot_prod(graph, xt_var)

    graph_cur_y = net(graph.reshape(-1, graph.shape[-1])).reshape(bn, -1, args.dout)
    yt_var_cur = net(xt_var)
    y_dot_prod = get_graph_dot_prod(graph_cur_y, yt_var_cur)

    return ((x_dot_prod - y_dot_prod) ** 2).mean()


def uniform_loss(x, t=2):
    return torch.pdist(x, p=2).pow(2).mul(-t).exp().mean().log()


def angular_optimize(xt, xv, xq, net, args, lambda_uniform, lambda_triplet, lambda_ang,
val_k, graph, knn, k_pos, k_neg,
                     valid, dfl):
    lr_schedule = [float(x.rstrip().lstrip()) for x in args.lr_schedule.split(",")]
    assert args.epochs % len(lr_schedule) == 0
    lr_schedule = repeat(lr_schedule, args.epochs // len(lr_schedule))
    print("Lr schedule", lr_schedule)

    n = xt.shape[0]
    xt_var = torch.from_numpy(xt).to(args.device)
    acc =[]
    valid_char = ""
    if len(valid) > 0:
        valid_char = "v"

    # prepare optimizer
    optimizer = optim.SGD(net.parameters(), lr_schedule[0], momentum=args.momentum)
    pdist = nn.PairwiseDistance(2)

    all_logs = []
    for epoch in range(args.epochs):
        # Update learning rate
        args.lr = lr_schedule[epoch]
        for param_group in optimizer.param_groups:
            param_group['lr'] = args.lr

        t0 = time.time()

        # Sample positives for triplet
        rank_pos = np.random.choice(k_pos, size=n)
        positive_idx = knn[np.arange(n), rank_pos]
        # positive_idx = gt_nn[np.arange(n), rank_pos]

        # Sample negatives for triplet
        net.eval()
        print("  Forward pass")
        xl_net = forward_pass(net, xt, 1024)
        print("  Distances")
        I = get_nearestneighbors_partly(xl_net, xl_net, k_neg, args.device, bs=3*10**5,
needs_exact=False)
        negative_idx = I[:, -1]

        # training pass
        print("  Train")
        net.train()
        avg_ang, avg_uniform, avg_loss = 0, 0, 0
        offending = idx_batch = 0
        avg_triplet = 0

        # process dataset in a random order
        perm = np.random.permutation(n)

        t1 = time.time()

        for i0 in range(0, n, args.batch_size):
            i1 = min(i0 + args.batch_size, n)
            bn = i1 - i0
            data_idx = perm[i0:i1]

            # anchor, positives, negatives
            xt_cur = xt_var[data_idx]
            pos = xt_var[positive_idx[data_idx]]
            neg = xt_var[negative_idx[data_idx]]

            # do the forward pass (+ record gradients)
            yt_cur = net(xt_cur)
            pos, neg = net(pos), net(neg)

            # triplet loss
            per_point_loss = pdist(yt_cur, pos) - pdist(yt_cur, neg)
            per_point_loss = F.relu(per_point_loss)
            loss_triplet = per_point_loss.mean()
            offending += torch.sum(per_point_loss.data > 0).item()

            # # entropy loss
            # I = pairwise_NNs_inner(ins.data)
            # distances = pdist(ins, ins[I])
            # loss_uniform_2 = - torch.log(distances).mean()

            # angular loss
            graph_cur = xt_var[graph[data_idx]]
            loss_ang = angular_loss(graph_cur,  xt_var[data_idx], net, bn, args)

            # uniform
            loss_uniform = uniform_loss(yt_cur)

            # combined loss
            loss = lambda_ang * loss_ang + lambda_uniform * loss_uniform + lambda_triplet
* loss_triplet

            # collect some stats
            avg_ang += loss_ang.data.item()
            avg_triplet += loss_triplet.data.item()
            avg_uniform += loss_uniform.data.item()
            avg_loss += loss.data.item()

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            idx_batch += 1

        avg_ang /= idx_batch
        avg_uniform /= idx_batch
        avg_triplet /= idx_batch
        avg_loss /= idx_batch

        logs = {
            'epoch': epoch,
            'loss_ang': avg_ang,
            'loss_uniform': avg_uniform,
            'loss_triplet': avg_triplet,
            'loss': avg_loss,
            'offending': offending,
            'lr': args.lr
        }

        t2 = time.time()

        if (epoch + 1) % args.val_freq == 0 or epoch == args.epochs - 1:
            logs_val = validation_function(net, xt, xv, xq, args, val_k)
            logs.update(logs_val)

        t3 = time.time()

        print ('epoch %d, times: [hn %.2f s epoch %.2f s val %.2f s]'
               ' lr = %f'
               ' loss = %g = %g + lam_u * %g + lam_tr * %g, offending %d' % (
            epoch, t1 - t0, t2 - t1, t3 - t2,
            args.lr,
            avg_loss, avg_ang, avg_uniform, avg_triplet, offending
        ))

        logs['times'] = (t1 - t0, t2 - t1, t3 - t2)
        all_logs.append(logs)

        if args.val_freq_search > 0 and ((epoch + 1) % args.val_freq_search == 0
or epoch == args.epochs - 1):
            import wrap.c_support as c_support
            net.eval()
            dim = xt.shape[1]
            yt = forward_pass(net, xt, 1024)
            knn_low_path = "/mnt/data/shekhale/models/nns_graphs/" + args.database +\
                           "/knn_1k_" + net_style + valid + ".ivecs"
            get_nearestneighbors_partly(yt, yt, 1000, args.device, bs=3 * 10 ** 5)
            save_transformed_data(xt, net, args.database + "/" + args.database
            if len(valid) > 0:
            acc_cur = round(acc_cur, 5)
            if args.save_optimal > 0:
            acc.append(acc_cur)
            net.train()
            print("Acc list ", acc)
            logs['acc'] = acc

  return all_logs
```

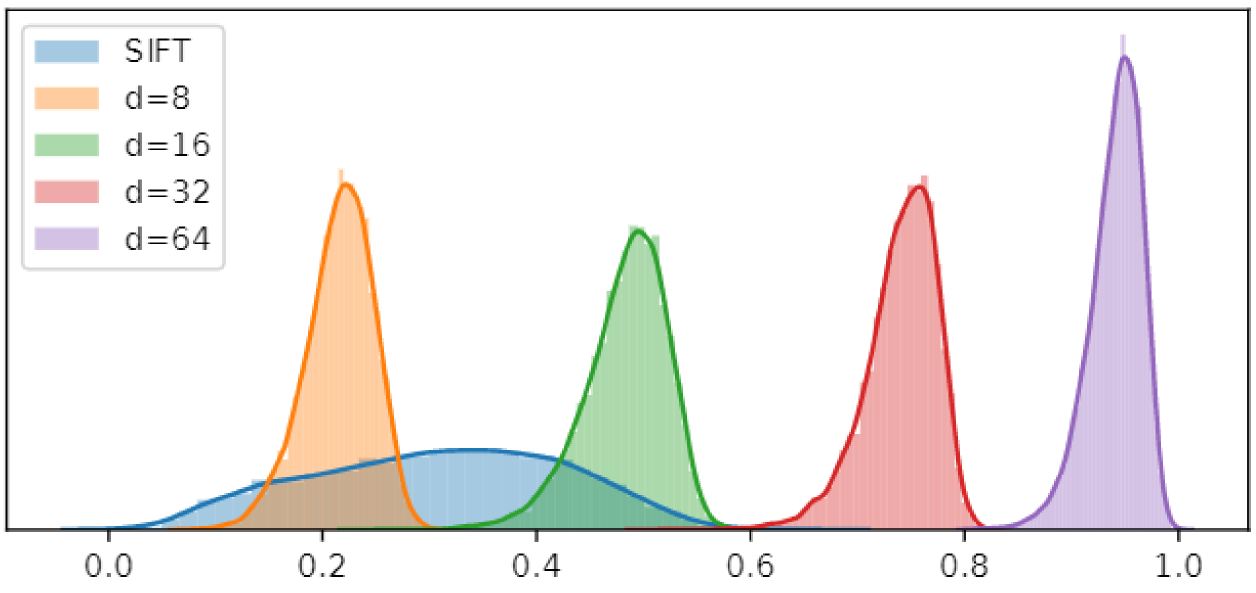
F. Python code for mapping the synthetic data and construct graphs by having random long range links so that the diameter of the graph reduces and we can converge at the point of the nearest neighbour in less time.

```python
def forward_pass(net, xall, bs=128, device=None):
    if device is None:
        device = next(net.parameters()).device
    xl_net = []
    net.eval()
    for i0 in range(0, xall.shape[0], bs):
        x = torch.from_numpy(xall[i0:i0 + bs])
        x = x.to(device)

        res = net(x)
        xl_net.append(res.data.cpu().numpy())

    return np.vstack(xl_net)


def forward_pass_enc(enc, xall, bs=128, device=None):
    if device is None:
        device = next(enc.parameters()).device
    xl_net = []
    enc.eval()
    for i0 in range(0, xall.shape[0], bs):
        x = torch.from_numpy(xall[i0:i0 + bs])
        x = x.to(device)
        res, _ = enc(x)
        xl_net.append(res.data.cpu().numpy())

    return np.vstack(xl_net)


def save_transformed_data(ds, model, path, device, enc=False):
    # ds = torch.from_numpy(ds).to(device)

    if enc:
        xb_var = torch.from_numpy(ds).to(device)
        xb_var = xb_var / xb_var.norm(dim=-1, keepdim=True)
        ds = xb_var.detach().cpu().numpy()
        del xb_var
    # ds = forward_pass_model(model, ds, 1024, lat=True)
    if enc:
        ds = forward_pass_enc(model, ds, 1024)
    else:
        ds = forward_pass(model, ds, 1024)
    # file_for_write_base = "data/" + path
    file_for_write_base = "/mnt/data/shekhale/data/" + path
    write_fvecs(file_for_write_base, ds)


def loss_permutation(x, y, args, k, size=10**4):
    perm = np.random.permutation(x.shape[0])
    k_nn_x = get_nearestneighbors(x[perm[:size]], x, k, args.device, needs_exact=True)
    k_nn_y = get_nearestneighbors(y[perm[:size]], y, k, args.device, needs_exact=True)
    perm_coeff = calc_permutation(k_nn_x, k_nn_y, k)
    print('top %d permutation is %.3f' % (k, perm_coeff))

    return perm_coeff


def loss_top_1_in_lat_top_k(xs, x, ys, y, args, kx, ky, size, name, fake_args=False):
    if xs.shape[0] != ys.shape[0]:
        print("wrong data")
    perm = np.random.permutation(xs.shape[0])
    top1_x = get_nearestneighbors(xs[perm[:size]], x, kx, args.device, needs_exact=True)
        top_neg_y  =  get_nearestneighbors(ys[perm[:size]],  y,  ky,  args.device,
needs_exact=True)
    ans_in_top_neg = 0
    for i in range(top1_x.shape[0]):
        if top1_x[i, -1] in top_neg_y[i]:
            ans_in_top_neg += 1
      print('%s:  Part  of  top1_x  in  gt_lat_  %d  =  %.4f'  %  (name,  ky,  ans_in_top_neg  /
len(top1_x)))
    return ans_in_top_neg / top1_x.shape[0]
```

Results

As a result of Dimensionality Reduction and creating graphs from data using the long range links and pre-sampled data points the difference can be understood in terms of the results on the number of queries per second vs error.





A. C++ and Python code for implementing Beam Search on graphs for NN.

```
#include <random>
#include <iostream>
#include <iomanip>
#include <fstream>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <unordered_map>
#include <unordered_set>
#include <map>
#include <cmath>
#include <ctime>
#include <queue>
#include <vector>
#include <omp.h>

#include <limits>
#include <sys/time.h>


#include <set>
#include <algorithm>
#include <ctime>

#include "search_function.h"

using  namespace std;

int main(int argc, char **argv) {

    size_t d_v = 3;
    if (argc == 2) {
        d_v = atoi(argv[1]);
    } else {
        cout << " Need to specify parameters" << endl;
        return 1;
    }

    time_t start, end;
    const size_t n = 1000000;  // number of points in base set
    const size_t n_q = 10000;  // number of points in query set
    const size_t n_tr = 1;
    const size_t d = d_v;  // dimension of data
    const size_t kl_size = 15; // KL graph size

    size_t knn_size = 100;
    size_t knn_size_for_beam = 20;
    if (d == 3) {
        knn_size = 20;
        knn_size_for_beam = 8;
    } else if (d == 5) {
        knn_size = 60;
        knn_size_for_beam = 10;
    } else if (d == 9) {
        knn_size = 300;
        knn_size_for_beam = 16;
    } else if (d == 17) {
        knn_size = 2000;
    }

    LikeL2Metric ll2 = LikeL2Metric();

    cout << "d = " << d << ", kl_size = " << kl_size << ", knn_size = " << knn_size <<
endl;

    std::mt19937 random_gen;
    std::random_device device;
    random_gen.seed(device());

    string path_data = "data/synthetic/synthetic";
    string path_models = "models/synthetic/";

    string dir_d = path_data + "_database_n_10_6_d_" + to_string(d) + ".fvecs";
    const char *database_dir = dir_d.c_str();  // path to data
    string dir_q = path_data + "_query_n_10_4_d_" + to_string(d) + ".fvecs";
    const char *query_dir = dir_q.c_str();  // path to data
    string dir_t = path_data + "_groundtruth_n_10_4_d_" + to_string(d) + ".ivecs";
    const char *truth_dir = dir_t.c_str();  // path to data


    string dir_knn = path_models + "knn_n_10_6_d_" + to_string(d) + ".ivecs";
    const char *edge_knn_dir = dir_knn.c_str();
```

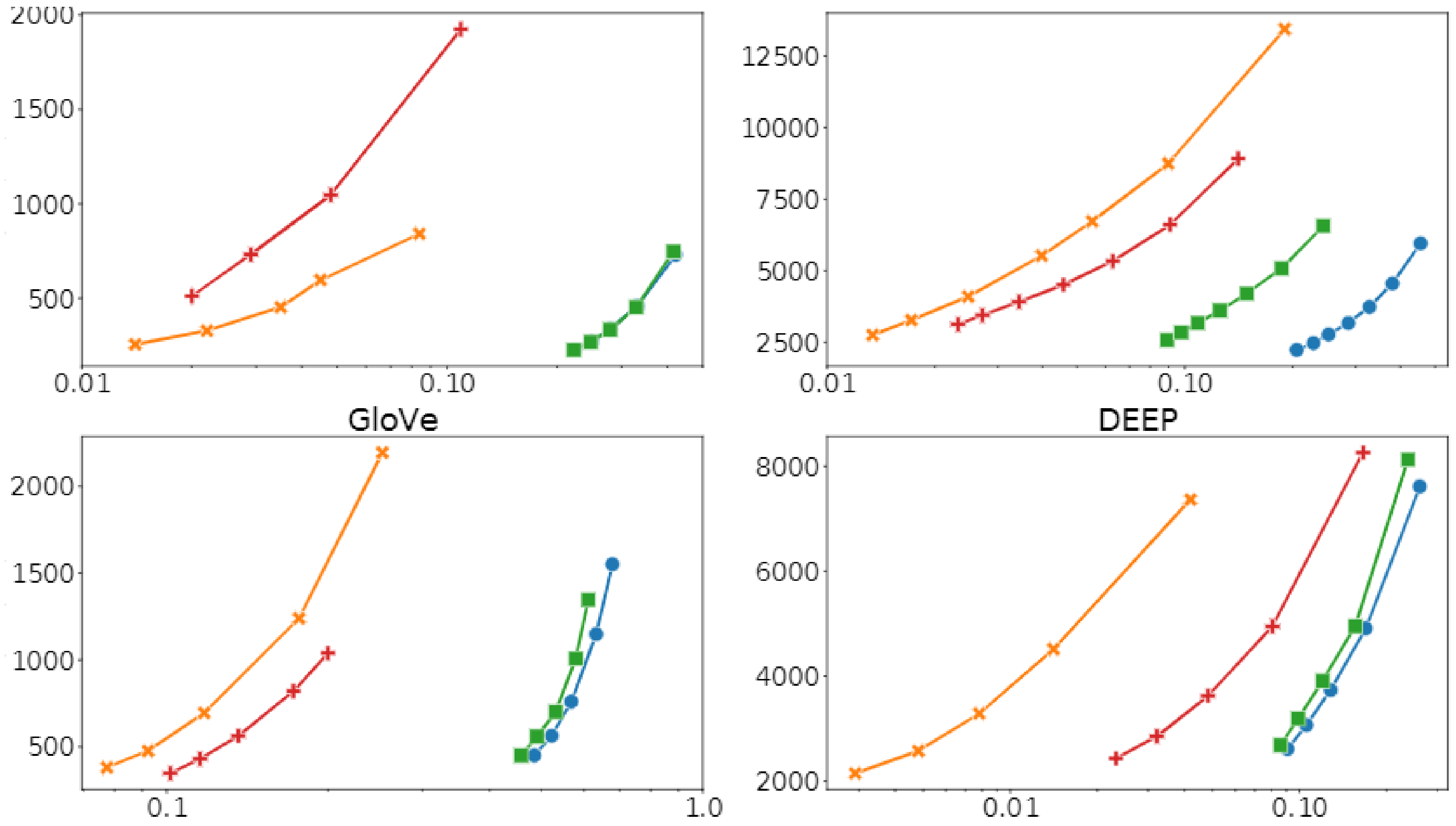
```
string dir_kl = path_models + "kl_sqrt_style_n_10_6_d_" + to_string(d) + ".ivecs";
const char *edge_kl_dir = dir_kl.c_str();


string output = "results/synthetic_n_10_6_d_" + to_string(d) + ".txt";
const char *output_txt = output.c_str();

remove(output_txt);


bool data_exist = FileExist(dir_d);
if (data_exist != true) {
    std::cout << "Creating data" << std::endl;
    vector<float> data = create_uniform_data(n_q + n, d, random_gen);
    vector<float> queries;
    for (int i=0; i < n_q*d; ++i) {
        queries.push_back(data[i]);
    }
    vector<float> db;
    for (int i=0; i < n*d; ++i) {
        db.push_back(data[n_q*d + i]);
    }
    vector<uint32_t> truth = get_truth(db, queries, n, d, n_q, &l12);

    std::ofstream data_input_db(database_dir, std::ios::binary);
    writeXvec<float>(data_input_db, db.data(), d, n);

    std::ofstream data_input_q(query_dir, std::ios::binary);
    writeXvec<float>(data_input_q, queries.data(), d, n_q);

    std::ofstream data_input_g(truth_dir, std::ios::binary);
    writeXvec<uint32_t>(data_input_g, truth.data(), n_tr, n_q);
}


std::cout << "Loading data from " << database_dir << std::endl;
std::vector<float> db(n * d);
{
    std::ifstream data_input(database_dir, std::ios::binary);
    readXvec<float>(data_input, db.data(), d, n);
}

std::vector<float> queries(n_q * d);
```

```
    {
        std::ifstream data_input(query_dir, std::ios::binary);
        readXvec<float>(data_input, queries.data(), d, n_q);
    }

    std::vector<uint32_t> truth(n_q * n_tr);
    {
        std::ifstream data_input(truth_dir, std::ios::binary);
        readXvec<uint32_t>(data_input, truth.data(), n_tr, n_q);
    }

    bool knn_exist = FileExist(dir_knn);
    if (knn_exist != true) {
        time(&start);
        ExactKNN knn;
        knn.Build(knn_size, db, n, d, &l12);
        cout << knn.matrixNN[0].size() << ' ' << knn.matrixNN[3].size() << endl;
        time(&end);
        cout << difftime(end, start) << endl;
        write_edges(edge_knn_dir, knn.matrixNN);
    }

    vector< vector <uint32_t>> knn(n);
    knn = load_edges(edge_knn_dir, knn);
    cout << "knn " << FindGraphAverageDegree(knn) << endl;
    knn = CutKNNbyK(knn, db, knn_size, n, d, &l12);
    cout << "knn " << FindGraphAverageDegree(knn) << endl;

    vector< vector <uint32_t>> knn_for_beam = CutKNNbyK(knn, db, knn_size_for_beam, n, d,
&l12);
    cout << "knn_for_beam " << FindGraphAverageDegree(knn_for_beam) << endl;

    bool kl_exist = FileExist(dir_kl);
    if (kl_exist != true) {
        time(&start);
        KLgraph kl_sqrt;
        kl_sqrt.BuildByNumberCustom(kl_size, db, n, d, pow(n, 0.5), random_gen, &l12);
        time (&end);
        cout << difftime(end, start) << endl;
        write_edges(edge_kl_dir, kl_sqrt.longmatrixNN);
    }

    vector< vector <uint32_t>> kl(n);
    kl = load_edges(edge_kl_dir, kl);
    cout << "kl " << FindGraphAverageDegree(kl) << endl;
    return 0;
}
```

BEAM SEARCH

```
#include <mutex>
#include <string.h>
#include <deque>

typedef uint16_t vl_type;

class VisitedList {
public:
    vl_type curV;
    vl_type *mass;
    size_t numelements;

    VisitedList(size_t numelements1)
    {
        curV = -1;
        numelements = numelements1;
        mass = new vl_type[numelements];
    }

    void reset()
    {
        curV++;
        if (curV == 0) {
            memset(mass, 0, sizeof(vl_type) * numelements);
            curV++;
        }
    };

    ~VisitedList() { delete mass; }
};
class VisitedListPool {
    std::deque<VisitedList *> pool;
    std::mutex poolguard;
    size_t maxpools;
    size_t numelements;

public:
    VisitedListPool(size_t initmaxpools, size_t numelements1)
    {
        numelements = numelements1;
        for (size_t i = 0; i < initmaxpools; i++)
            pool.push_front(new VisitedList(numelements));
    }

    VisitedList *getFreeVisitedList()
    {
        VisitedList *rez;
        {
            std::unique_lock<std::mutex> lock(poolguard);
            if (pool.size() > 0) {
                rez = pool.front();
                pool.pop_front();
            }
            else {
                rez = new VisitedList(numelements);
            }
        }
        rez->reset();
        return rez;
    };

    void releaseVisitedList(VisitedList *vl)
    {
        std::unique_lock<std::mutex> lock(poolguard);
        pool.push_front(vl);
    };

    ~VisitedListPool()
    {
        while (pool.size()) {
            VisitedList *rez = pool.front();
            pool.pop_front();
            delete rez;
        }
    };
};
```

B. We used .ipynb Jupyter Notebook for printing out the graph of different rates of QPS vs the Error induced due to the number of iterations.



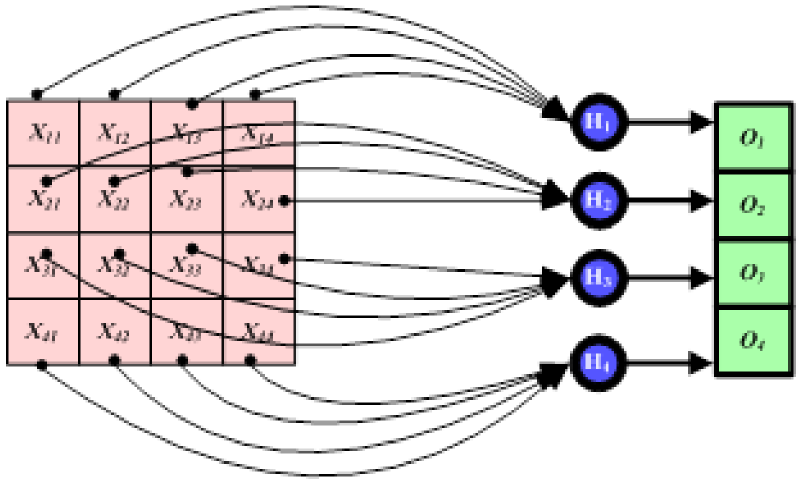
Therefore we can deduce that after using Beam Search and Long Range Links when applied on real dataset mapped onto unit Euclidean sphere acts somewhat as Synthetic data and the number of queries that can be served increase hence the time to query one data point decreased hence we can now find the Nearest Neighbour effectively.

Conclusions

In this project we applied the NNS techniques coupled with Beam Search and Long Range Links to find the nearest neighbours in Real Datasets by mapping and creating the Graph of relation of real dataset on a unit Euclidean Sphere. We expect that when graph-based NNS algorithms become more popular, more theoretical study of such approaches will follow. Finding broader circumstances under which comparable assurances may be provided would be a sensible path for further study. In particular, we assume uniform distribution in the current study. We found empirical evidence for this hypothesis, although a more comprehensive examination of more generic distributions would be beneficial. While our findings may be easily extended to distributions that are similar to uniform, further extensions appear to be difficult. Analyzing the effect of diversification is another potential area.

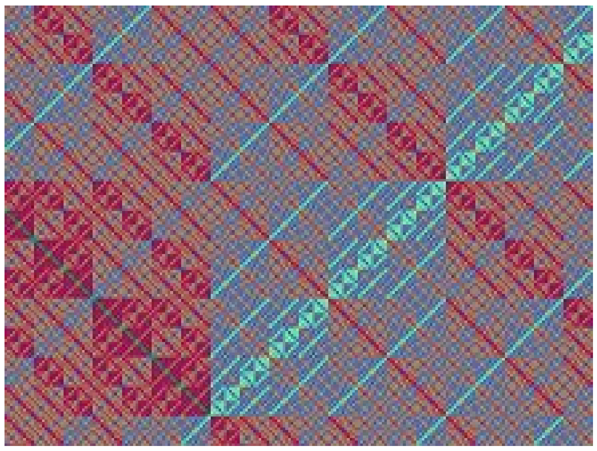
Applications

The Nearest Neighbour Search method that we coded can be used for carrying out general NN search within a metric space in following areas.

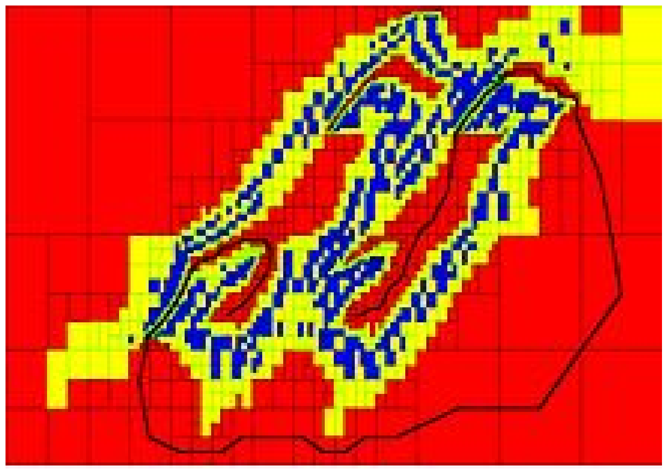


X_{11} X_{12} X_{13} X_{14}
X_{21} X_{22} X_{23} X_{24}
X_{31} X_{32} X_{33} X_{34}
X_{41} X_{42} X_{43} X_{44}
H_1 H_2 H_3 H_4
O_1 O_2 O_3 O_4

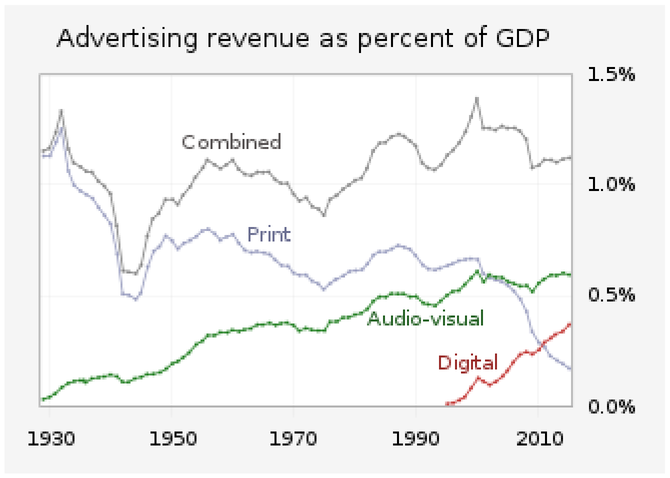
Pattern Recognition



Coding Theory



Sampling based motion planning



Internet Marketing

References

1. Reproducing Synthetic Experiments Paper from International Conference on Machine Learning 2020. [Link]

2. Optimal Data Dependent Hashing for approximate near neighbours in Graph from In Proceedings of the 47th Annual ACM Symposium on Theory of Computing and Graphs. [Link]

3. ANN-benchmarks: a benchmarking tool for approximate nearest neighbor algorithms ,Information Systems, 2019. [Link]

4. Locally Sensitive Hashing in Graphs. [Link]

5. The research on accuracy optimization of beam. [Link]

6. Randomized partition trees for nearest neighbor search Research Paper. [Link]

7. Implementation of Beam Search Decoder for a Graph. [Link]

8. Study On Dimensionality Reduction Techniques And Applications. [Link]